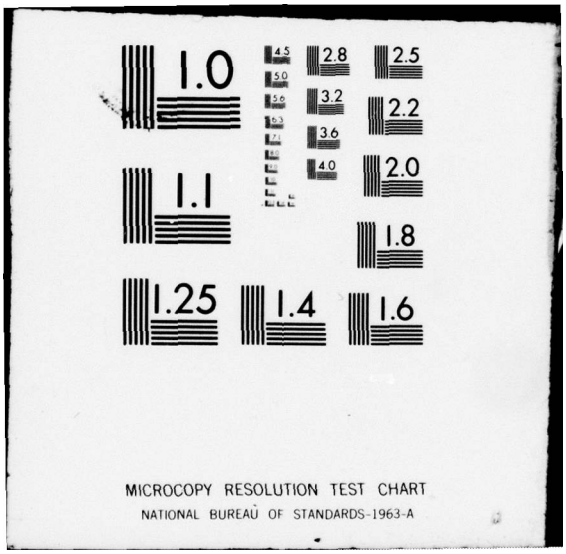
AD-A070 957 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
TESTING DATA ABSTRACTIONS THROUGH THEIR IMPLEMENTATIONS.(U)
MAY 79 R HAMLET, M ARDIS, J GANNON AFOSR-77-3181
UNCLASSIFIED TR-761 AFOSR-TR-79-0796 NL

1 OF 1
AD
A070957

END
DATE
FILMED
8-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9
10
11
12
13
14
15
16
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LEVEL II

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND
20742

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR-79-0796	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): TESTING DATA ABSTRACTIONS THROUGH THEIR IMPLEMENTATIONS	5. TYPE OF REPORT & PERIOD COVERED: Interim rept.
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Richard Hamlet, John Gannon, Mark Ardis, Paul McMullin	8. CONTRACT OR GRANT NUMBER(s): AFOSR-77-3181
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science, University of Maryland, College Park, Maryland 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, D.C. 20332	12. REPORT DATE: May 1979 (12) 40 P. / 13. NUMBER OF PAGES: 37
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (16) 2304 (17) A2	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited. (14) TR-761

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A data abstraction can be specified by a syntactic (domainrange) description of its objects and operations on them, plus a set of axioms describing the behavior of the operations. In a programming language that supports abstraction, abstract objects are given a representation in terms of built-in data types (or other, previously defined abstractions), and the abstract operations are defined as procedures. The implemented abstraction has a meaning, acquired from the definition of the programming language; the axioms also specify an independent meaning. The crucial question is then whether or not either meaning is (continued)

DD FORM 1 JAN 73 1473

409 022

UNCLASSIFIED

20. Abstract (continued)

"correct," that is, whether it corresponds to the desired abstraction that the programmer had in mind. Using a finite collection of tests, the axiom meaning and the code meaning can be compared for consistency. A compiler-based system, DAISTS, is described for automating this process.

The work reported here was supported by a grant from the Air Force Office of Scientific Research (AFOSR 3181B). Computer time was provided in part by the Computer Science Center of the University of Maryland.

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Technical Report TR-761 May 1979

AFOSR-77-3181

TESTING DATA ABSTRACTIONS THROUGH THEIR IMPLEMENTATIONS

Richard Hamlet
Mark Ardis
John Gannon
Paul McMullin

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

A data abstraction can be specified by a syntactic (domain-range) description of its objects and operations on them, plus a set of axioms describing the behavior of the operations. In a programming language that supports abstraction, abstract objects are given a representation in terms of built-in data types (or other, previously defined abstractions), and the abstract operations are defined as procedures. The implemented abstraction has a meaning, acquired from the definition of the programming language; the axioms also specify an independent meaning. The crucial question is then whether or not either meaning is "correct," that is, whether it corresponds to the desired abstraction that the programmer had in mind. Using a finite collection of tests, the axiom meaning and the code meaning can be compared for consistency. A compiler-based system, DAISTS, is described for automating this process.

The work reported here was supported by a grant from the Air Force Office of Scientific Research [redacted]. Computer time was provided in part by the Computer Science Center of the University of Maryland.

0. Introduction

The formal basis for data abstractions began with their implementation in SIMULA classes [Dahl et al. 1970], and Hoare's correctness-based definition [Hoare 1972] that has now become known as the "abstract model" approach, refined into a program-proving method by the Alphard group [Wulf et al. 1976]. The "pure algebraic" approach [ADJ 1978, Guttag 1977, Zilles 1975] avoids both assertion-based correctness and implementation. Algebraic axioms may themselves be used as rewriting rules to execute trial expressions from a data type. The OBJ [Goguen and Tardo 1979] and AFFIRM [Musser 1979] systems use this idea to make specifications "executable" without a conventional program doing the computations. In these systems there is no data-abstraction programming language, only a specification language in which sample terms can be evaluated by symbolic methods.

Our bias is toward practical use of abstractions, as implemented in the programming language SIMPL-D. This view probably began with the MIT CLU abstraction project [Liskov 1976]. However, a data-abstraction language by itself faces the problem of communicating what the implementations mean. The language nicely gives the syntax of the abstract objects, but their semantics must be conveyed by commentary or by reading the code. The former is unreliable, and the latter violates the very reason for encapsulating abstractions in the first place. Also, without an independent expression of the intended meaning, we are unable to judge mechanically the success of the implementing code. We thus decided that the usefulness of SIMPL-D would be enhanced by the addition of specifications in a machine-processable form. The bias toward practical programming remains, however, so we decided to keep away from correctness ideas in favor of testing ideas. Our plan is to use algebraic-equation specifications, and to test whether or not implementing code meets them.

An immediate difficulty is that such a view has no theoretical underpinning--the pure-algebraic approach does not consider independent implementation in a programming language, and the abstract-model approach is intertwined with correctness methods. We must develop definitions appropriate to our situation, definitions in which the implementation is primary, and "correctness" is an idea that a human being has, unconnected with proof formalism. The resulting definition can be viewed as that of Hoare, with the assertion-correctness removed. We consider it a unification of the algebraic and abstract-model approaches, but from the algebraic side, unlike the effort of [Flon and Misra 1979].

This report describes the abstraction programming language SIMPL-D (Section 1); defines the meaning of "correct implementation" and "specification by axioms" (Section 2); gives language and compiler extensions needed to support testing the consistency of axioms and code (defining DAISTS, Section 3); and, explores the significance of successful tests (Section 4).

1. SIMPL-D

SIMPL-D is a member of the SIMPL family of programming languages [Basili 1976] with features that permit the declaration of abstract data types. This section first describes the basic features that are common to several members of the SIMPL family of languages and then discusses the features that are unique to SIMPL-D.

SIMPL-D shares the following language features with some of the other languages in the SIMPL family.

1. A program is a series of global variable declarations followed by a series of procedure declarations. One procedure is designated as the starting point of execution by including its name in a START command as the last line of a program.

2. There is no block structure and procedures may not be nested. Each procedure may access global variables, formal parameters, and local variables.

3. Procedures may be recursive. Procedures may not be passed as parameters; scalar data objects may be passed by value or reference and aggregates are passed by reference. Only scalar objects may be returned as values.

4. The language is strongly typed with the basic data types integer, character, and string. Explicit conversion routines exist to coerce values from one of the basic types to another. Zero is false and nonzero is true in contexts requiring boolean values (e.g., IF statements). The usual arithmetic, relational, and logical operators are available. In addition, there are string operators for concatenation and substring selection.

5. The only intrinsic type generator is the array of single dimension with compile-time determinable bounds. The lower bound for arrays is fixed at zero.

6. The statements of the language include assignment, IF,

CASE, WHILE, EXIT, CALL, and RETURN. There are primitives for stream and record input and output.

7. Separate compilation is supported using ENTRY and EXTernal declarations.

The primary new feature of SIMPL-D is the CLASS. CLASS declarations define new types which may subsequently be used in other declarations. The interior of a CLASS is a series of variable declarations (the representation) followed by a series of procedure declarations (the body). Either the representation (by declaring a CLASS to be CLEAR) or the procedures of the body (by listing them in the CLASS heading) may be visible outside a CLASS declaration. A CLASS with no procedures associated with it and whose representation is available to users is simply a record. Conversely, a CLASS whose representation can only be accessed by users through the procedures of the CLASS defines an abstract data type.

Users generally view CLASS objects as indivisible entities. Inside the CLASS, these objects may be viewed as a sequence of declarations of more primitive objects. The primitive objects may be declared with one of the storage attributes UNIQUE or SHARED.

```
CLASS Stack = Push, Pop, Top, Empty

   /* representation */

   SHARED INT ARRAY Values(1000)
   SHARED INT ARRAY Nextvalue(1000)
   SHARED INT Avail
   UNIQUE INT Stacktop

   /* body */

   .
   .
   .
ENDCLASS
```

UNIQUE components of a CLASS are allocated each time a CLASS object is created. SHARED components of a CLASS are allocated only once (no matter how many CLASS objects are created) at program initiation and are common to all objects of the same type. In the example above, each Stack object consists of a single integer component, Stacktop. The value of a Stack object might be constructed by using its Stacktop component as a reference to a linked list whose links are stored in Nextvalue. The links in Nextvalue reference positions in the integer array Values that contain the values currently the list. All Stack objects share the common space for values and links; no Stack object need overflow until the storage requirements for all Stack objects exceeds 1000 values.

The body of a CLASS is a series of procedure and function declarations. If one of these routines accepts a formal parameter of the CLASS being defined, the body of the routine may access the UNIQUE components of the formal using a period notation similar to that of Pascal or PL/I. Since only a single copy of the SHARED components exist, no similar qualification is necessary to access them. Continuing our example from above, we might define Pop as follows:

```
Stack FUNC Pop(Stack S)

   INT Temp

   IF Empty(S)
      THEN
         WRITE('*** underflow ***')
         ABORT
      END

   /* put location of previous value in Stacktop */
   Temp := S.Stacktop
   S.Stacktop := Nextvalue(Temp)

   /* update chain of available locations */
   Nextvalue(Temp) := Avail
   Avail := Temp

   RETURN(S)
```

A list of integer parameters to the CLASS may be used to control the sizes of the UNIQUE components of the representation. These parameters are themselves treated as UNIQUE components of an object, but their values may only be altered by the assignment of an object to the entire object of which they are a component.

```
CLASS Stack(Stacksize) = Push, Pop, Top, Empty

   UNIQUE INT ARRAY values(Stacksize)
   UNIQUE INT Stacktop
   .
   .
   .
ENDCLASS
```

The appearance of the reserved word ASSIGN in the operation list enables the SIMPL assignment operation (:=) to be applied to objects of this CLASS type. As is the case with objects having primitive SIMPL types, applying the assignment operation to CLASS objects results in the value of the right operand being copied to the location of the left operand. If the run-time value of the right operand is too large to store in the left operand, an error results. Assignment is not allowed for classes containing SHARED components because of possible side effects on shared components.

Two nameless procedures are available to provide automatic initialization of UNIQUE and SHARED data of a CLASS. SHARED components are initialized once at program initiation through an implicit call to the appropriate nameless procedure. UNIQUE objects are initialized at the time of their creation (either program initiation for globals or procedure invocation for locals) through an implicit call to the appropriate nameless procedure. The CLASS defining the type Stack above might contain such a procedure to initialize Stack objects to be empty.

```
INIT UNIQUE PROC(Stack S)
   S.Stacktop := -1
```

Declarations establish the types and sizes of variables, but the size of a variable is not part of its type. Size parameters

cannot be specified in the declarations of formal parameters or function results; these objects have sizes that cannot be determined until run time. The size values of formal parameters are those of the corresponding actual parameters. The size value of a function result is that of the expression appearing in the return statement. Thus objects of arbitrary sizes can be passed to, and returned from, routines. In the following example, S1 and S2 are variables with the same type (Stack), but different sizes (50 and 5 respectively).

```
Stack(50) S1
Stack(5) S2
.
.
.
S1 := Pop(S2)
```

The following is a full definition of the data type "bounded" Stack and its use in a SIMPL-D program. The operations available on Stack objects are listed in the CLASS heading; the code implementing these operations (*except for* ASSIGN which is system-defined) is found in the CLASS body. Push places a new value on top of a Stack object and returns the object as the result of the operation. Attempts to place a new value on a full Stack object are ignored. Pop removes a value from a nonempty Stack object and has no affect on empty Stack objects. Top returns the value on top of a Stack object, but does not remove the value from the object. Top returns Undefined (In this example, Undefined is 0 as a result of the macro definitions controlling the representation.) if it is applied to an empty Stack object. NewStack returns as its value an empty Stack object. Depth can be used to discover the number of values in a Stack object and Limit gives the upper bound on the number of values that can be placed in any Stack object. An array of EltType (defined as integer by a macro) values and an index of the last value, StackTop, form the representation for Stack objects. Each time a Stack object is bound to storage, space is reserved for the array and integer index.

```
CLASS Stack = Push, Pop, Top, Empty, NewStack,
              StackEqual, Depth, Limit, ASSIGN

/* macro definitions to control representation */
DEFINE EltType = 'INT'
DEFINE Undefined = '0'
DEFINE StackSize = '20'

/* representation */
UNIQUE EltType ARRAY Values(StackSize) /* 0..StackSize-1 */
UNIQUE INT StackTop

/* body containing operation definitions */

Stack FUNC NewStack
   Stack Result
   Result.Stacktop := -1
   RETURN(Result)

Stack FUNC Push(Stack S, EltType Elt)
   Stack Result
   IF S.StackTop + 1 = StackSize
      THEN
         RETURN(S) /* return stack unchanged */
      END
   Result := S
   Result.StackTop := Result.StackTop + 1
   Result.Values(Result.StackTop) := Elt
   RETURN(Result)

Stack FUNC Pop(Stack S)
   Stack Result
   IF Empty(S)
      THEN
         RETURN(S)
      END
   Result := S
   Result.StackTop := Result.StackTop - 1
   RETURN(Result)

EltType FUNC Top(Stack S)
   IF Empty(S)
      THEN
         RETURN(Undefined)
      END
   RETURN(S.Values(S.StackTop))

Bool FUNC Empty(Stack S)
   RETURN(S.StackTop = -1)
```

```
Bool FUNC StackEqual(Stack P, Stack Q)
   INT I
   IF Depth(P) = Depth(Q)
      THEN
         I := Depth(P)
         WHILE I > 0 DO /* compare all elements */
            IF P.Values(I) <> Q.Values(I)
               THEN
                  RETURN(False)
               END
            I := I - 1
            END
         RETURN(True)
      END
   RETURN(False)

INT FUNC Depth(Stack S)
      RETURN(S.StackTop + 1)

INT FUNC Limit
      RETURN(StackSize)

ENDCLASS /* end of declaration of CLASS Stack */


PROC P /* program using variables of type Stack */

   INT I
   Stack S
   S := NewStack

   /* read and add values to S */
   WHILE .NOT. EOI DO /* EOI is end-of-input */
      READ(I)
      S := Push(S,I)
      END

   /* write and remove values from S */
   WHILE .NOT. Empty(S) DO
      WRITE(Top(S),SKIP)
      S := Pop(S)
      END

START P
```

2. Correct Implementation of Data Abstractions

A language like SIMPL-D supporting data abstraction defines abstractions using a collection of its primitive objects and procedures. Because the language has a formal semantic definition, these elements have an intuitive meaning, roughly that the primitive objects correspond to some abstract objects (*e.g.*, strings over some finite alphabet, or natural numbers), and the procedures to mappings among the abstract objects. An implementation of a data abstraction therefore has meaning, constructed from the language-definition meanings of the objects and procedures used. The question is whether this meaning corresponds to what a human being had in mind for the abstract object. We begin by framing a simple definition that captures this idea.

Suppose that the programming language has but a single primitive type, whose meaning is a set of abstract objects D. (In the usual "structured programming" treatment [Linger *et al.* 1979] the objects D are the integers, the meanings of program variables of type *int*.) The meanings of procedures are mappings of cross products of D into cross products of D. (To obtain a tuple of outputs it may be necessary to resort to "output parameters"; in a language like ALGOL 68 such objects may be function values directly.) The semantics of a programming language can be thought of as the definition of the meaning mapping: an assignment of the abstract function to the procedure fragment of the language syntax. Because it is important to know when such an abstract function is under discussion, we name them with a peculiar notation due to Kleene: the meaning function of the program fragment P is written $[P]$. Thus for a function procedure P with two parameters,

$$[P]: D \times D \longrightarrow D$$

is the meaning. The semantic definition of the language establishes the detailed correspondence between P and $[P]$.

we say that P computes [P] . The crucial point is that by using the language definition we can discuss not P , but [P] , and thus deal only with abstract, intuitive, meaning objects, not concrete, syntactic ones.

In this discussion we have not committed to whether or not the semantic domain D is finite. In any actual implementation it must be, but there are good reasons to take it to be in principle unbounded as Turing did. In fact, data abstraction itself can be used to bridge the gap, because potentially unlimited objects like strings can be implemented as character arrays using dynamic memory allocation, the ultimate limitation being only the exhaustion of virtual memory.

The domain of a data abstraction is like D , except that it does not correspond to any built-in programming-language objects. For simplicity let it be a single kind of abstract objects, say the set A . Let the objects D play the dual role of meaning for programming language primitive objects, and a second abstract domain for opposition to A . The abstract *operations* are mappings from cross products of A and D into A or D . (The complication of permitting cross product ranges here adds nothing.) The abstract operations also have no built-in language procedures. The data abstraction features of the language provide a means of defining language-to-abstraction correspondences. In the simple case we are considering, the abstract objects A are made to correspond to tuples from D , and abstract operations to procedures mapping these tuples. That is, the programmer implementing an abstraction mentally establishes a representation mapping R ,

$$R: D^{k} \longrightarrow A \quad \text{(onto)}$$

carrying the built-in meanings to the desired ones. This mapping is reflected in the language by an ordered collection of declarations. In SIMPL-D these are the declarations of a CLASS in order, so that

```
CLASS C = ...

UNIQUE INT x, y, z

...
```

would establish the representation as having a domain of triples of (abstract) integers. Note that in this view the implementation syntax defines the abstraction domain, so that no abstract syntax is needed in addition; however, the object such a triple is to represent is entirely unspecified.

Similarly, the data abstraction facilities of the language allow definition of language procedures that correspond to the abstract operations, and the language syntax defines the abstract domains and ranges. For example, in SIMPL-D if there were just one operation carrying a pair of abstract objects into the language-primitive objects, we would have

```
CLASS C = op

UNIQUE INT x, y, z

INT FUNC op(C a, C b)
...
```

in which the CLASS name C is used to stand in for the representing triple in the parameter list. Again the syntax carries over to the abstraction, but there is no meaning given there.

Because of the way this syntax is constructed, it is impossible to write a syntactically correct SIMPL-D program that does not implement some abstraction, so we make this the first definition: an implementation of a data abstraction is just a representation and a collection of function declarations. The implementation defines the syntax of the abstract object and abstract operations.

Once programming language code has been filled in for the functions declared in an implementation of an abstraction, one

meaning is defined, by that implementation through the language's semantic definition. But the programmer is supposed to have an abstract meaning in mind *a priori*, so we now define what it means for this intuitive idea to be captured by the code. It is clear that we should assert that the given program-induced meaning is the one intended. For example, in the case of a representation

$$R: \quad D \times D \longrightarrow A$$

(that is, pairs), and an abstract operation

$$f: \quad A \times D \longrightarrow A$$

implemented by a procedure P (in SIMPL-D it would appear as

```
CLASS A = P
UNIQUE INT x, y
A FUNC P(A u, INT z)
...
```

if the primitive type D is INT), the code represented by the body of P defines [P] as a mapping

$$[P]: D^2 \times D \longrightarrow D^2 .$$

Correctness of the implementation then amounts to the assertion that the following diagram commutes:

$$\begin{array}{ccc} & & f \\ & A \times D & \longrightarrow A \\ R \uparrow & & \uparrow R \\ & [P] & \\ & D^2 \times D & \longrightarrow D^2 \end{array}$$

we call this an implementation diagram of P . In general, the implementation is correct iff each of its functions satisfies this condition; namely that any representation tuple, carried into the abstract A by R and mapped by the abstract

operation, produces the same (abstract) value as mapping that representation tuple by the implementing procedure, then representing the result. In symbols, for the example, correctness requires that for any pair $(s,t) \in D^2$, and any $x \in D$,

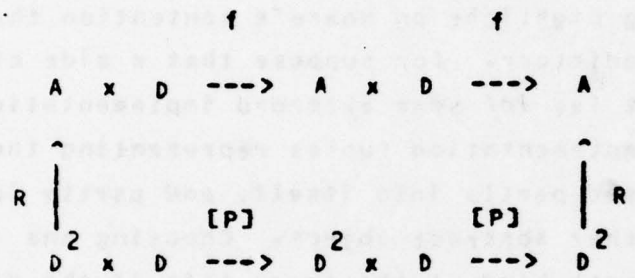
$$f(R(s,t), x) = R([P](s, t, x)) .$$

(In the special case that D itself is a part of the abstract domain, we require that the identity representation be used, and we do not show it in the implementation diagram.)

We emphasize again that this definition is framed entirely in the abstractions D, A, f, and $[P]$. Of course, there is a correspondence with the language object P through $[P]$, and the tuples from D have language correspondents, but these do not occur in the definition. Note also that the abstract syntax comes entirely from the implementation syntax--it has no independent existence, and the abstract meaning is so far confined to the mental processes of human beings.

Side effects of the implementation can also be discussed in this context. Intuitively, the implementation has side effects if executing its procedures not only yields their returned values, but furthermore alters the state of some internal storage. In SIMPL-D, "internal storage" is the SHARED and UNIQUE data of a CLASS, which we are treating as a tuple of objects from the set D. In framing a definition of "side effect," should <u>all</u> storage within the implementation be in the domain of the intuitive representation function, or should some of it (say the SHARED part) be in a hidden, "side-effect" store? We choose to include everything in the representation domain, because it preserves the intuition that this mapping captures the entire correspondence between implementation objects and abstract ones, and because the resulting mathematics is cleaner. (As an example of the latter, the implementation diagrams explicitly show all the relevant data.)

Intuitively, there is a side effect if storage is different after a procedure has been invoked than it was before. To capture the idea of a time history, we consider an <u>extended implementation diagram</u> that displays a succession of implementation procedure invocations. For example, for the operation $f: A \times D \to A$ above, the implementing P might be used on its own output, leading to the diagram:



(In the diagram we omit not only the identity map $D \to D$, but the internal $R: D^2 \to A$.)

Formally, there is a <u>side effect</u> in a sequence of implementation procedures iff an extended implementation diagram exists with the following property: Starting at the upper left, some abstract x is the image under the representation R of the set (of tuples)

$$E = \{d \mid R(d) = x\} .$$

Some $x_0 \in E$ is carried along the bottom of the diagram back into E by the implementation procedures in composition. However, there exists a $y \in E$ that is carried along the bottom of the diagram to y' distinct (as a tuple) from y. A side effect is <u>benevolent</u> iff every such $y' \in E$.

The intuition behind this definition (following [Hoare 1972]) is that "side effect" is meaningless unless the extended implementation diagram returns to a representation of the same abstract object. That is, the notion depends not just on the implementing code, but on the representation that the programmer

had in mind. So long as extended implementation diagrams produce sequences of representations of <u>different</u> abstract objects, the effect is primary, not "side". When there is a return to the same a ra t o e t, but with a different representing tuple, a side ef ct has occurred. Benevolence means that such changes do not matter--the representation mapping washes them out. The somewhat surprising thing is that in any abstraction correctly implemented by given code, any side effects <u>must</u> be benevolent, hrowing an interesting highlight on Hoare's contention that the terms are almost contradictory. For suppose that a side effect is not benevolent; that is, for some extended implementation diagram a set E of implementation tuples representing the same abstract object is mapped partly into itself, and partly into representations of another abstract object. Choosing one representing tuple of each kind at the lower left of the diagram, on the upper (abstract) path they lead to the same object, but on the lower (implementation) path they do not. Hence the implementation cannot be correct.

When the representation mapping is 1-1, side effects are impossible accor ing to this definition. There are no classes of representing tuples other than singletons, and so the benevolent confusion within such a class cannot occur. The usual notion of a "side effect" that is <u>not</u> benevolent is formally just an incorrect implementation. f the object y that is sent to $y' \neq y$ by the code is one that displays a side effect, and procedure Q misbehaves on y' where it would not have misbehaved on y, there are two possible cases according to the formal definition:

(1) The side effect in the extended diagram prior to the pplication of Q is benevolent; that is, the representation does n t distinguish between y and y'. Then Q itself is incorrect.

(2) The side effect is not benovelent; that is, the representation maps y and y' onto different abstract objects.

Then somewhere in the extended diagram prior to the application of Q there is an error in the implementation.

The situation regarding side effects can be summarized as follows: correctness of an implementation (as defined by commuting implementation diagrams for all its procedures) guarantees that all extended implementation diagrams will also commute.

According to the definition above, the implementation fixes a syntax--the sets of objects, and the names, domains, and ranges of operations. This syntax is called the signature of the abstraction, as in [ADJ 1978]. We now investigate the structure of all abstractions with a fixed signature. For any two abstractions A_1 and A_2 with the same signature, a homomorphism $h: A_1 \rightarrow A_2$ satisfies equations such as

$$h(f(x,y)) = f(h(x),h(y))$$

for all x, y in the domain of f in A_1, for all operations f of the signature (with appropriate argument lists--here shown as pairs). We use the same name for the operation f in both abstractions. An onto homomorphism is an epimorphism; a 1-1 epimorphism is an isomorphism. We say that A_2 is a homomorphic (epimorphic, isomorphic) image of A_1 (under h). Identifying all isomorphic abstractions, define a partial order by $A_1 \subseteq A_2$ iff A_1 is an epimorphic image of A_2. The trivial abstraction $\emptyset$ containing one element mapped to itself by all operations is the least element in this partial ordering.

When a person sets out to implement an intuitive abstraction in a programming language like SIMPL-D, he proceeds by first choosing a representation, then writing code to manipulate its tuples appropriately. However, given a SIMPL-D CLASS definition,

it may implement certain abstractions correctly, whatever the programmer had in mind. One candidate is always the trivial abstraction. In that case, whatever the programmer writes for code is correct if the procedures do not abort or fail to terminate, because every tuple is mapped to the same abstract point. If the trivial abstraction is not what the programmer had in mind, it is tempting to say that the representation is incorrect, but according to our definition that distinction cannot be made--the range of the representation mapping is entirely intuitive. Another intuitive abstraction always correctly implemented by any code is one that mirrors the implementation itself. That is, the representation is an identity map, and the intuitive operation corresponding to procedure P is just [P] . In this case the top and bottom lines in the implementation diagram are identical, so it necessarily commutes. (Here even the possible failures of termination in the code are of no consequence.)

Between the trivial abstraction and the one that mirrors the implementation, all elements of the partial ordering are correctly implemented, because given any A that is an image of the mirror-image abstraction under a homomorphism h, h itself is a representation onto A that forces the appropriate diagrams to commute. It is likely that the successful programmer had some element of this partial order in mind, but probably not the extreme elements. Insofar as the implementation takes advantage of the full detail of the representing tuples, the abstractions it correctly implements are likely to cluster near the implementation-mirror end of the order; if the representing structure is largely ignored, near the trivial end.

The definition above captures implementation of an *a priori* abstract data type. If the human being who envisioned the type is available to act as an oracle, and can perform the representation function, then the definition can be used directly to perform tests of the implementation. The person chooses a

representing tuple, maps it to the corresponding abstract object, performs abstract operations and records the result. The implementing program is given the same representing tuple and produces another such tuple; the human being also maps this to its corresponding abstraction and compares it with the recorded result. Agreement means a successful test. This process cannot be automated without some form of specification that describes the abstract type independent of the implementation. We have chosen algebraic axioms [Guttag 1977] as the most attractive such specification method. The exact form the axioms take can be left open; in particular, we have not decided the several questions of (a) whether or not the functions involved must be total (and hence the form recursions are allowed to take), (b) whether or not existential quantifiers will be allowed as well as universal quantifiers, and (c) whether of not "conditional" cases will be allowed in the axioms. Our testing scheme will work in any combination of decisions about these matters, so the decisions can be deferred.

Whatever form the axioms take, they are a finite collection of relations among terms constructed from abstract functions, closed by quantifying the variables. An intuitive abstraction satisfies a set of axioms iff each is true in the natural interpretation into the intuitive domain (in the technical logic sense). That is, the intuitive axiom relation holds of the intuitive term objects. Given any point in the partial order of abstractions satisfying a set of axioms, points below it in the partial order also satisfy them, because the epimorphism that maps downward in the partial order commutes with all the operations, and this is sufficient to show that the axioms are inherited. We report more results on the relationship between the implemented abstractions and the specified abstractions in [Ardis and Hamlet 1979].

Given a SIMPL-D CLASS and a set of axioms written in terms of its operation names, there are thus three things of interest

within the partial ordering of abstractions:

(1) The abstraction the programmer had in mind.

(2) The set of abstractions that the code correctly implements.

(3) The set of abstractions that satisfy the axioms.

If it happens that (1) is somewhere in (2) and also in (3), the programmer can be said to have succeeded. (However, more could be demanded. Some of those taking the algebraic-axiom approach often require that (1) correspond to the initial element of (3), viewed as a category [ADJ 1978]. We do not insist on this additional condition, although we would be pleased to be able to detect that it is satisfied.) There are many ways for the programmer to fail, however. Since (1) does not and cannot appear explicitly in the SIMPL-D text, technically we can only investigate the relationship between (2) and (3). However, both code and axioms are human attempts to capture (1), and they are sufficiently different that we may hope their relationship will illuminate their common source, even though it is not formally in evidence.

Testing is the mechanism we have chosen to investigate the relationship between axioms and code. Test points in the form of implementation tuples can be inserted in the axioms viewed as expressions in SIMPL-D. This amounts to associating with each axiom its extended implementation diagram, and moving across the bottom of the diagram. Since the outermost function in each axiom is equality, and since the representation mapping into the Boolean domain {true, false} is likely to be one-one, we can view the test as supplying half the information about commutativity for one point.

Let us suppose that such a test fails. That is, some representation tuples are chosen, the appropriate values that occur in an axiom are worked out (in implementation tuples

again), and the resulting value represents false. Then if the axiom does hold for the intuitive abstract objects, the implementation must be incorrect (in the technical sense above), and its failure lies in one or more of the procedures whose names appear in the axiom. Since the axiom holds of the abstract objects, the test point must lead around the upper path of the extended implementation diagram to true; but, the lower path leads to false. This can occur only if one of the component implementation diagrams similarly fails to commute. It is important to note that nothing is assumed about the specification "capturing" the intuitive abstraction except that a single axiom is true of it.

There is an exception when an axiom uses existential quantification. Then the additional possibility exists that the code is not incorrect, but rather the data points used are so sparse that no representation of the needed object was included. Furthermore, an existential failure is over a whole set of points, and not for a single point, thus making it harder to trace. Perhaps these are sufficient reasons to avoid existential quantifiers in axioms.

Unfortunately, a successful test gives less information: if the axioms define the abstraction, and the implementation satisfies them for a point, it could yet happen that several of the implemented funcions are not correct for the point, but in combination their errors cancel each other to make the test succeed. The case of an existentially quantified axiom is again noteworthy: the existence of a tuple that satisfies the axiom may be illusory if the procedures implementing the operations on it are incorrect. We do not expect tests to establish the correctness of the implementation, but the success of a test when the code is wrong for the test inputs is particularly disturbing. This difficulty can occur even with exhaustive testing, and even if the equality function of the abstraction is correctly implemented. Intuitively, it arises because the definition of

correctness (following Hoare) proceeds function-by-function, while the algebraic-equation specification may only define the operations in combination. Of course, the success of an axiom involving only one operation, under exhaustive testing and assuming a correct implementation of equality, is definitive.

We do not contemplate exhaustive testing, but we do want to know that an incorrect implementation possesses a test point that exposes its error. With the coverage criteria described in Section 4, we can then measure the liklihood that a given set of tests include the crucial points. If all tests might succeed, yet errors remain, our testing scheme is on shaky ground.

5. DAISIS: SIMPL-D with Axioms and Tests

Axioms and test cases can be appended to a SIMPL-D CLASS as defined in Section 1. The form is:

```
    :
    : SIMPL-D CLASS definition
    :
AXIOMS
    :
    : algebraic axioms describing the abstraction
    :
TESTPOINTS
    :
    : declaration and initialization of test values
    :
TESTSETS
    :
    : sets of test points to be used with each axiom
    :
START
```

we call the resulting system DAISTS for exactly what it is: Data Abstraction Implementation, Specification, and Testing System.

In the AXIOMS section each axiom has a name, an optional free variable list containing the types and names of each free variable used in the axiom body, a colon, an axiom body, and a terminating semicolon. The axiom body has the form:

```
<left side> = <right side>
```

where either side may contain references to free variables and operations. In addition, the <right side> may contain a conditional expression like that of ALGOL 60:

```
IF <boolean> THEN <exp> ELSE <exp>
```

One of the familiar axioms of the bounded stack CLASS appears as:

```
Pop2(Stack S, EltType I):
   Pop(Push(S,I)) = IF Depth(S) = Limit
                    THEN Pop(S)
                    ELSE S;
```

This axiom tells us that the result of popping a Stack S after pushing a new value I onto S is the same as either:

1. the value of the object obtained by popping S (If S already contained the maximum number of values before the push, attempts to push a new value on S would be ignored.), or

2. the value of the original Stack S otherwise.

The TESTPOINTS section looks like a SIMPL-D procedure, complete with local declarations and executable statements. This section has been included to allow users to build objects to be referenced in the TESTSETS section of the program. An object that is expensive to construct can thus be used in testing several axioms without repeating its construction. For example, a useful point for testing a bounded Stack might be obtained as:

```
Stack S1
S1 := Push(Push(Push(NewStack,1),2),3)
```

The TESTSETS section of the program contains a list of axiom names with values to be substituted for the free variables of the axioms. For example, a three-element test set for the Pop2 axiom is:

```
Pop2: (NewStack,1), (S1,2), (S1,3);
```

This set of tests and the Pop2 axiom will be used to "exercise" the implementation. First with NewStack bound to S and 1 bound to I, we will use the body of the Pop2 axiom as a driver program that invokes implementation functions. If the left and right sides of an axiom do not return values that are judged to be equal by the appropriate implementation function (here, StackEqual), a diagnostic message is printed indicating that the axiom has failed. This process is repeated with S1 (an object initialized in the TESTPOINTS section) bound to S and 2 bound to I, and finally with S1 bound to S and 3 bound to I.

In order to achieve this behavior, we compile the axioms into procedures and the test sets into calls on the appropriate procedures. The Pop2 axiom becomes:

```
PROC Pop2(Stack S, EltType I)
   IF .NOT. StackEqual(Pop(Push(S,I),
                IF Depth(S)=Limit THEN Pop(S) ELSE S)
      THEN
         WRITE('*** axiom Pop2 failed ***')
      END
```

and the test set for Pop2 is translated into the following code:

```
CALL Pop2(NewStack,1)
CALL Pop2(S1,2)
CALL Pop2(S2,3)
```

The following program is the bounded stack example from Section 1 augmented with axioms and test sets. Rather than repeat the original CLASS declaration for the bounded stack type, we have employed the SIMPL-D USE directive which obtains the CLASS's representation and operation interface information from the file appearing as its arguement. This file must have been previously created by a separate compilation of the module containing the bounded stack declaration. We have reproduced the interface information in the comments following the USE directive. Notice that the axioms are parameterized by the type EltType, which is defined by macro definitions preceding the axioms. However, the test sets are given in terms of integer values, the type bound to EltType during this compilation. No test sets are given for the axioms Empty1, Top1, Pop1, and Depth1. These axioms involve no free variables; the single calls to test them are generated automatically.

```
/+ USE 'Stack' +/

/* operations for Stack */
/* Stack FUNC Push(Stack, INT) */
/* Stack FUNC Pop(Stack) */
/* INT FUNC Top(Stack) */
/* INT FUNC Empty(Stack) */
/* INT FUNC StackEqual(Stack,Stack) */
/* INT FUNC Depth(Stack) */
/* INT FUNC Limit */

DEFINE True = '1', False = '0',
       EltType = 'INT', Undefined = '0'
```

```
AXIOMS /* the "functional" model */

Empty1:
    Empty(NewStack) = True;

Empty2(Stack S, EltType I):
    Empty(Push(S,I)) = False;

Top1:
    Top(NewStack) = Undefined;

Top2(Stack S, EltType I):
    Top(Push(S,I)) = IF Depth(S) = Limit
                     THEN Top(S)
                     ELSE I;

Pop1:
    Pop(NewStack) = NewStack;

Pop2(Stack S, EltType I):
    Pop(Push(S,I)) = IF Depth(S) = Limit
                     THEN Pop(S)
                     ELSE S;

Depth1:
    Depth(NewStack) = 0;

Depth2(Stack S, EltType I):
    Depth(Push(S,I)) = IF Depth(S) = Limit
                       THEN Depth(S)
                       ELSE Depth(S) + 1;

Equal0:
    StackEqual(NewStack,NewStack) = True;

Equal1(Stack P, Stack Q, EltType I, EltType J):
    StackEqual(Push(P,I),(Push(Q,J)) =
        IF Depth(P) < Limit
        THEN
            IF Depth(Q) < Limit
            THEN
                IF I = J
                THEN StackEqual(P,Q)
                ELSE False
            ELSE StackEqual(Push(P,I),Q)
        ELSE
            IF Depth(Q) < Limit
            THEN StackEqual(P,Push(Q,J))
            ELSE StackEqual(P,Q);

Equal2(Stack P,Stack Q):
    StackEqual(P,Q) = StackEqual(Q,P);

Equal3(Stack P, EltType I):
    StackEqual(Push(P,I),NewStack) = False;
```

```
TESTPOINTS /* to be used in the test sets */

   Stack S1, S2
   INT I

   S1 := Push(Push(Push(NewStack,1),2),3)
   S2 := S1
   I := 4
   WHILE I <= Limit DO
      S2 := push(S2,I)
      I := I + 1
      END


TESTSETS /* to "test" the axioms */

   Empty2: (S1,7), (NewStack,1);
   Top2: (NewStack,3), (S2,2);
   Pop2: (NewStack,1), (S1,2), (S2,3);
   Depth2: (NewStack,1), (S1,2), (S2,3);
   Equal1: (S1,S1,2,3), (S1,S1,2,2), (S2,S2,1,2),
           (S2,S2,2,2), (S1,S2,4,5), (S1,S2,3,3);
   Equal2: (NewStack,S1), (S1,NewStack), (S1,S2), (S2,S1);
   Equal3: (NewStack,7), (S2,2);


START
```

The very first running version of DAISTS uncovered an error in the implementation of Depth. The Depth1 axiom was translated into:

```
   IF .NOT. (Depth(NewStack) = 0)
      THEN
         WRITE('*** error in axiom Depth1 ***')
      END
```

Our original Depth function was implemented as follows:

```
   INT FUNC Depth(Stack S)
         RETURN(S.StackTop)
```

(compare Section 1). The StackTop component of a Stack object is assigned the value -1 in NewStack to indicate that the object is empty. (The index range of SIMPL arrays is 0..upperbound-1.) Thus, Depth(NewStack) returned -1 and the message that the test data failed the Depth1 axiom was printed.

The axioms and tests cannot guarantee that the implementation is without errors. Errors in the implementation may interact in ways which still satisfy the axioms. For example, if Push simply

updated the StackTop component of Stack objects without retaining any values, the only axiom to fail would be Top2 since Top(Push(S,I)) ≠ I. Now, if the equality function for EltType objects was also incorrect (always returning true), the axioms would all be satisfied by a faulty implementation. However, if we require the user to supply enough test data to cover every part of his axioms, he may find that he cannot supply test data to the Equal1 axiom to reach the ELSE clause guarded by the condition:

Depth(P) < Limit & Depth(Q) < Limit & I ≠ J

because I = J. Preliminary ideas about coverage monitoring are presented in the next section.

4. Testing, Structural Constraints, and Correctness

In conventional program testing, specifications enter in the form of an "oracle," a means of magically obtaining the correct results with which the test is to agree. In practice this oracle is a human being examining the output, perhaps too often willing to agree with the program. (This position can be justified--some programs are solving problems that can be attacked no other way.) It is crucial to distinguish an oracle that can judge the output for *any* input, from a "limited oracle"--a set of given input-output pairs encoding only a part of the complete behavior. The latter can be obtained by physical simulation and by hand calculation, but when structural testing criteria are used, inputs are generated that may not keep within a given collection.

Testing based on pure input-output pairs has little to recommend it; since any test is finite, it could be met by a program written as a *case* statement covering the given pairs, and otherwise entering a never-ending loop. The addition of structural test criteria, additional constraints on the collection of tests, has its origin in avoiding such counterexamples. The simplest structural criterion is that each and every statement of the tested program must be executed for some input among the tests. If the every-statement criterion is met, then it cannot be that the program is hiding a special block of code that takes some unpleasant action outside the finite test collection. Put the other way, it is certainly wise to have used all the code in some way; entirely untried code can contain arbitrarily nasty bugs. Formally, structural criteria are designed to lead to *reliability* of a test--the property that if errors are not exposed, there are no errors. Let a program P be given the meaning (function computed) $[P]$ by its semantic definition, and let f be the function that the program ought to compute (according to the idea of some human being). Then a collection of test data D is *reliable for* P iff

$$[P]|_D = f|_D \implies [P] = f .$$

This definition (due to [Howden 1976]) makes an exhaustive test trivially reliable. Every program has a finite reliable test, but there is no way to judge mechanically a given program P and test data D to determine if D is reliable for P [Howden, 1976, Hamlet, 1977a]. By strengthening the specification to include more information about <u>how</u> the program must behave, or by restricting the set of errors that must be detected, mechanically checkable reliability can be attained [Hamlet 1978].

The DAISTS processor described in Section 3 provides a rich testing vehicle. The dual machine-processable texts of axioms and implementing code can make do without an oracle. When trying a test point in an axiom, we do not know what operation values should result, but we do know a relationship that should hold among them. The virtue in this form of oracular information is that it is available for any and all test points that might be tried, not only those of a preselected set. The drawbacks of spurious success have been noted in Sections 2 and 3. In DAISTS, tests can be required to satisfy structural criteria not only on the program code, but on the axioms as well. In this section we explore the potential of tests with structural constraints for detecting inconsistencies between abstractions the code correctly implements, and abstractions that satisfy the axioms. In the process we also hope to learn about the relationship of code and axioms to the abstraction the programmer had in mind.

Section 2 describes the significance of a test failure in an axiom compiled into code: either the implementation or the axiom (or both) is inconsistent with the intuitive abstraction desired. It is probably easiest for a person to analyze the code function-by-function, and turn to the axiom only if each step in its extended implementation diagram checks out. In what follows it is assumed that each error in code or axiom is corrected as discovered. The result of testing is then a collection of points

for which the axioms do execute successfully. Structural criteria are now introduced to enhance the significance of the successful test collection.

For conventional program code there are path and expression structural constraints, corresponding to the control/data dichotomy. A combination of the two can be reliable, but not within a time feasible for real programs. Hence, a number of approximations to the impractical exhaustive constraints exist. Monitoring a particular constraint can be accomplished by a run-time support routine in DAISTS (the path constraints are particularly easy to handle). In what follows we concentrate on the significance of constraints, not on details of approximations and monitoring algorithms, which are straightforward. The compiler-based testing approach [Hamlet, 1977b] has the virtue that all testing is accomplished by a run-time module, in which the algorithms are written in SIMPL-D, and therefore easy to change and to experiment with. We therefore expect to defer decisions about what and how to monitor as long as possible.

Path criteria are the easier to understand. All possible inputs to a program divide into equivalence classes according to which control-flow path each input follows. A given test collection meets an exhaustive path constraint iff it includes one member from each such equivalence class. The unreliability of path testing [Howden 1976] results from an improper or insufficient coverage of some path-defined class. Approximations to the exhaustive path constraint are necessary, because when code contains loops, the number of paths may be unbounded. The simplest approximation requires each statement to be executed for some test point.

Within axioms compiled into code, the only analog of path constraints occurs when there are conditional cases, and then exhaustive path testing is the same as "branch testing"--requiring that both alternatives be taken. Except for this case, the test points necessarily take the single path through each

axiom that corresponds to moving across the bottom of its extended implementation diagram. However, it may be helpful to the programmer to "cover" one axiom at a time, and observe the path coverage that results within the implementation procedures.

Expression constraints on code have a better axiom analog than do path containts. All possible inputs to a program are divided into equivalence classes by value sequences (histories) assigned to a single expression within the program. A given test collection meets an exhaustive expression constraint iff it includes one member from each such class. Expression constraints are less used in practical testing because they are harder to approximate, and even the approximations are expensive to monitor. One of the simplest approximations is that of forbidding constant subexpressions--a set of tests is lacking if any nonconstant part of any expression has exactly the same value history for all test points.

Since axioms are expressions, they are covered just as is code, according to an expression constraint. An obvious interpretation if axiom expressions remain constant, is that (insofar as the tests can determine) the trivial abstraction has been specified. Axiom expression coverage has more significance than coverage for code, just because there are few axiom "paths" to consider.

The interplay between structural constraints for axioms and for code suggests a testing methodology something like the following: Begin by finding test points that cover the axioms, with as little redundancy as possible. The deficiencies of such a test collection in code coverage indicate parts of the code to examine for errors (usually of the sort that the correctly implemented abstractions are too complex). If the code proves to be correct on examination, seek additional test points to cover it, and examine the redundant coverage these points introduce in the axioms, suggesting that the abstractions satisfying them are too simple.

5. Status of DAISTS

Work is proceeding on the DAISTS processor. At this writing, the processor will permit users to supply implementations, axioms, and test points and will "exercise" the implementation. In the next stage of development, we plan to add simple structural testing criteria to check statement coverage and expression variability.

References

[Ardis and Hamlet 1979]

M.A. Ardis and R.G. Hamlet. The structure of specifications and implementations of data abstractions. (in preparation).

[ADJ 1975]

J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Abstract data types as initial algebras and the correctness of data representations. Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structure, (May, 1975), 89-93.

[ADJ 1978]

J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. in R. Yeh (ed.), Current Trends in Programming Methodology 4, Prentice-Hall, Englewood Cliffs, N.J., (1978), 80-149.

[Basili 1976]

V.R. Basili. The design and implementation of a family of application-oriented languages. Proceedings of the Fifth Texas Conference on Computing Systems, (October 1976), 6-12.

[Dahl et al. 1970]

O.-J. Dahl, B. Myhrhaug, and K. Nygaard. The SIMULA 67 Common Base Language. Norwegian Computing Center, S-22, Oslo, (1970).

[Flon and Misra 1979]

L. Flon and J. Misra. A Unified Approach to the Specification and Verification of Abstract Data Types. Proceedings of Specifications of Reliable Software, (1979), 162-169.

[Goguen and Tardo 1979]

J.A. Goguen and J.J. Tardo. An introduction to OBJ: a language for writing and testing formal algebraic program specifications. Proceedings of Specifications of Reliable Software, (1979), 170-189.

[Guttag 1977]

J.V. Guttag. Abstract data types and the development of data structures. CACM 20, 6, (June 1977), 396-404.

[Hamlet 1977a]

R.G. Hamlet. Testing Programs with Finite Sets of Data. Computer Journal 20, (March 1977), 232-237.

[Hamlet 1977b]

R.G. Hamlet. Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering SE-3, (July 1977), 279-290.

[Hamlet 1978]

R.G. Hamlet. Critique of Reliability Theory. IEEE Workshop on Software Testing, Fort Lauderdale, Florida, (December 1978), 56-69.

[Hoare 1972]

C.A.R. Hoare. Proof of correctness of data represenations. Acta Informatica 1, (1972), 271-281.

[Howden 1976]

W.E. Howden. Reliability of the path analysis testing strategy. IEEE Transactions on Software Engineering SE-2, (September 1976), 208-215.

[Linger et al. 1979]

R. C. Linger, H. D. Mills, and B. I. Witt. Structured Programming Theory and Practice. Addison-Wesley, 1979.

[Liskov 1976]

B.H. Liskov. An introduction to CLU. M.I.T., Computation Structures Group, 136, (February 1976).

[Musser 1979]

D.R. Musser. Abstract data type specification in the Affirm system. Proceedings of Specifications of Reliable Software, (1979), 47-57.

[Wulf et al. 1976]

W.A. Wulf, R.L. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. IEEE Transactions on Software Engineering SE-2, (December 1976), 253-265.

[Zilles 1975]

S.N. Zilles. Algebraic specification of data types. Project MAC Progress Report for 1973-74, (MIT CSG Memo 119), 1-12.